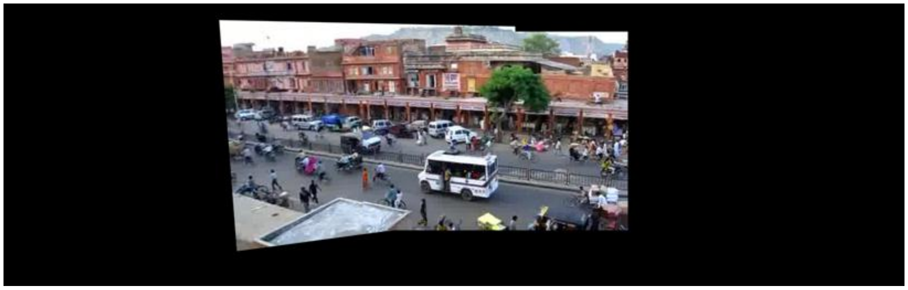
Programming Project #5: Video Stitching and Processing

CS445: Computational Photography - Fall 2019

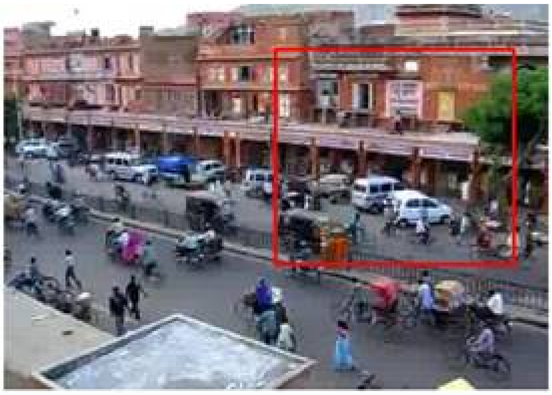
Part I: Stitch two key frames

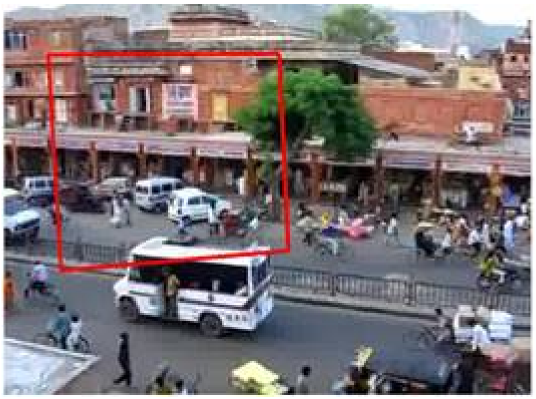


This involves:

1. compute homography H between two frames;
2. project each frame onto the same surface;
3. blend the surfaces.

Check that your homography is correct by plotting four points that form a square in frame 270 and their projections in each image, like this:





In [1]:

```
import cv2
import numpy as np
from numpy.linalg import svd, inv

%matplotlib inline
from matplotlib import pyplot as plt
```

In [2]:

```
pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
pts = pts.reshape((-1,1,2))
print(pts)
```

```
[[[10  5]]

 [[20 30]]

 [[70 20]]

 [[50 10]]]
```

In [3]:

```
# images location
im1 = './images/input/frames/f0001.jpg'
im2 = './images/input/frames/f0270.jpg'

# Load an color image in grayscale
im1 = cv2.imread(im1) # float32, 0.0-1.0
im2 = cv2.imread(im2)
print(im1.shape[0],im1.shape[1],im1.shape[2])
```

```
360 480 3
```

```
In [4]: def auto_homography(Ia,Ib, homography_func=None):
            '''
            Computes a homography that maps points from Ia to Ib

            Input: Ia and Ib are images
            Output: H is the homography

            '''
            if Ia.dtype == 'float32' and Ib.dtype == 'float32':
                Ia = (Ia*255).astype(np.uint8)
                Ib = (Ib*255).astype(np.uint8)

            Ia_gray = cv2.cvtColor(Ia,cv2.COLOR_BGR2GRAY)
            Ib_gray = cv2.cvtColor(Ib,cv2.COLOR_BGR2GRAY)

            # Initiate SIFT detector
            sift = cv2.xfeatures2d.SIFT_create()

            # find the keypoints and descriptors with SIFT
            # kp_a is keypoints List and des_a is descriptors List
            kp_a, des_a = sift.detectAndCompute(Ia_gray,None)
            kp_b, des_b = sift.detectAndCompute(Ib_gray,None)

            # BFMatcher with default params
            bf = cv2.BFMatcher()
            matches = bf.knnMatch(des_a,des_b, k=2)
        #     print('---TEST values in matches, which is list of elements(a,b)---')
        #     print(matches[0])
        #     print(matches[0][0].distance)
        #     print(matches[0][1].distance)

            # Apply ratio test
            good = []
            for m,n in matches:
                if m.distance < 0.75*n.distance:
                    good.append(m)

            numMatches = int(len(good)) # numMatches = 215
            matches = good

            # Xa and Xb are 3xN matrices that contain homogeneous coordinates for th
            # matching points for each image
            Xa = np.ones((3,numMatches))
            Xb = np.ones((3,numMatches))

        #     print('---TEST 3*215 point---')
        #     print(Xa[:,0][0:2])
        #     print(Xa[:,0])
            pts1_good = np.ones((3,4))
            pts2_good = np.ones((3,4))
            for idx, match_i in enumerate(matches):
                Xa[:,idx][0:2] = kp_a[match_i.queryIdx].pt
                Xb[:,idx][0:2] = kp_b[match_i.trainIdx].pt

            ## RANSAC
            niter = 1000
```

```
    best_score = 0

    H = np.zeros((3,3))
    for t in range(niter):
        # estimate homography
        subset = np.random.choice(numMatches, 4, replace=False)
        pts1 = Xa[:,subset]
        pts2 = Xb[:,subset]

        H_t = homography_func(pts1, pts2)
#         H_t = computeHomography(pts1, pts2) # edit helper code below (comp
        # pts1 = 3xN matrix and N = 4

        # score homography
        Xb_ = np.dot(H_t, Xa) # project points from first image to second us
        du = Xb_[0,:]/Xb_[2,:] - Xb[0,:]/Xb[2,:]
        dv = Xb_[1,:]/Xb_[2,:] - Xb[1,:]/Xb[2,:]

        ok_t = np.sqrt(du**2 + dv**2) < 1  # you may need to play with this

        score_t = sum(ok_t)

        if score_t > best_score:
            best_score = score_t
            H = H_t
            in_idx = ok_t
            pts1_good = pts1
            pts2_good = pts2
#     print(pts1_good)
#     print(pts2_good)
    print('best score: {:02f}'.format(best_score))

# # Check that your homography is correct by plotting four points that form
#     # use as the four corners to draw the polylines
#     con_pts1 = np.zeros((4,2),np.int32)
#     con_pts2 = np.zeros((4,2),np.int32)

#     for i in range (4):
# #         print("point %d Xa"%(i),pts1_good[:,i])
# #         print("point %d Xb"%(i),pts2_good[:,i])
# #         Xb_ = np.dot(H, Xa)
# #         print("point %d Xb_"%(i),Xb_[:,i])
# #         print(Xb_[:,i]/Xb_[2,i])
#         con_pts1[i] = pts1_good[0:2,i]
#         con_pts2[i] = pts2_good[0:2,i]
# #         print(con_pts1[i])
# #         print(con_pts2[i])
# #     A[[i, j], :] = A[[j, i], :] # 实现了第i行与第j行的互换
#     im1_RGB2 = cv2.cvtColor(Ia,cv2.COLOR_BGR2RGB)
#     im2_RGB2 = cv2.cvtColor(Ib,cv2.COLOR_BGR2RGB)
#     con_pts1[[0, 2], :] = con_pts1[[2, 0], :]
#     con_pts2[[0, 2], :] = con_pts2[[2, 0], :]

# #   for draw rectangular  im1_RGB2 = cv2.rectangle(im1_RGB2,(100,100),(20
# #     pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
#     con_pts1 = con_pts1.reshape((-1,1,2))
#     con_pts2 = con_pts2.reshape((-1,1,2))
# #     print("--------")
# #     print(con_pts1)
# #     print(con_pts2)
#     im1_RGB2 = cv2.polylines(im1_RGB2,[con_pts1],True,(255,0,0),thickness
#     im2_RGB2 = cv2.polylines(im2_RGB2,[con_pts2],True,(255,0,0),thickness

#     plt.imshow(im1_RGB2)
#     plt.show()

#     plt.imshow(im2_RGB2)
#     plt.show()

#     # Optionally, you may want to re-estimate H based on inliers

    return H
```

In [5]:

```
def computeHomography(pts1, pts2):
    '''
    Compute homography that maps from pts1 to pts2 using least squares solve

    Input: pts1 and pts2 are 3xN matrices for N points in homogeneous
    coordinates.  N is 4.

    Output: H is a 3x3 matrix, such that pts2~=H*pts1
    '''

    dimen, num = pts1.shape
    A = np.zeros((2*num, 9), dtype = np.float32)

    for i in range (0, num):
        # p_prime = w*[u_prime,v_prime,1]
        # p = [u,v,1]
        p_prime = pts2[:,i]
        u_prime = p_prime[0]/p_prime[2]
        v_prime = p_prime[1]/p_prime[2]

        p = pts1[:,i]
        u = p[0]/p[2]
        v = p[1]/p[2]
        A[i*2,:] = [-u,-v,-1,0,0,0,u*u_prime, v*u_prime, u_prime]
        A[i*2+1,:] = [0,0,0,-u,-v,-1,u*v_prime, v*v_prime, v_prime]
    U,S,Vt = svd(A)
    h = Vt[-1,:]
    H = np.reshape(h, (3, 3))

#     std1_u = np.std(pts1[1,:])
#     std1_v =  np.std(pts1[2,:])
#     mean1_u = np.mean(pts1[1,:])
#     mean1_v = np.mean(pts1[2,:])
#     T1 = np.matrix([[1/std1_u, 0, 0],[0, 1/std1_v, 0],[0, 0, 1]]) * np.mat

#     std2_u = np.std(pts2[1,:])
#     std2_v = np.std(pts2[2,:])
#     mean2_u = np.mean(pts2[1,:])
#     mean2_v = np.mean(pts2[2,:])
#     T2 = np.matrix([[1/std2_u, 0, 0],[0, 1/std2_v, 0],[0, 0, 1]]) * np.mat
#     H = T2/H_n * T1
    return H
```

In [6]:

```
H = auto_homography(im1,im2, computeHomography)
```

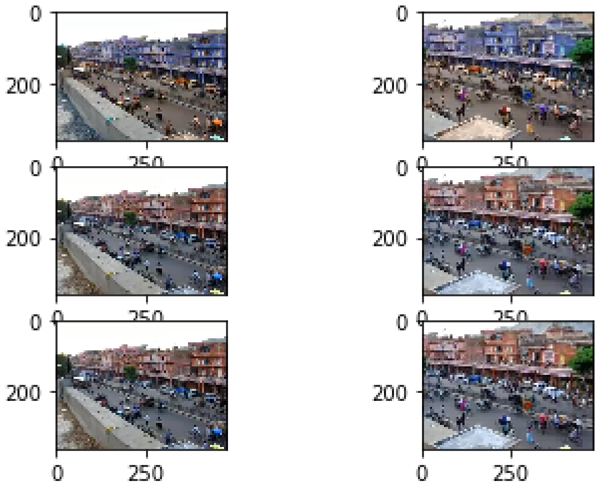
```
best score: 144.000000
```

In [7]:

```
im1_path = './images/input/frames/f0001.jpg'
im2_path = './images/input/frames/f0270.jpg'
im1_cv2 = cv2.imread(im1_path) # uint 0-255
im2_cv2 = cv2.imread(im2_path)

im1_RGB = cv2.cvtColor(im1_cv2,cv2.COLOR_BGR2RGB)
im2_RGB = cv2.cvtColor(im2_cv2,cv2.COLOR_BGR2RGB)

im1_plt = im1_RGB.astype(np.float32)/255
im2_plt = im2_RGB.astype(np.float32)/255


# im1 = cv2.imread(im1_path,cv2.COLOR_BGR2RGB)
# im2 = cv2.imread(im2_path,cv2.COLOR_BGR2RGB)
# Load an color image in grayscale
# im1 = plt.imread(im1) # float32, 0.0-1.0
# im2 = plt.imread(im2)

fig, axes = plt.subplots(3,2)
axes[0,0].imshow(im1_cv2)
axes[0,1].imshow(im2_cv2)
axes[1,0].imshow(im1_RGB)
axes[1,1].imshow(im2_RGB)
axes[2,0].imshow(im1_plt)
axes[2,1].imshow(im2_plt)
"It is important to change color from BGR to RGB"
```

Out[7]:

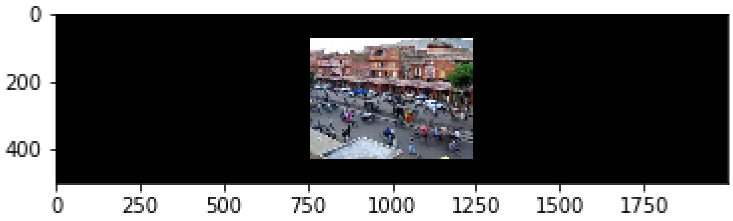
```
'It is important to change color from BGR to RGB'
```

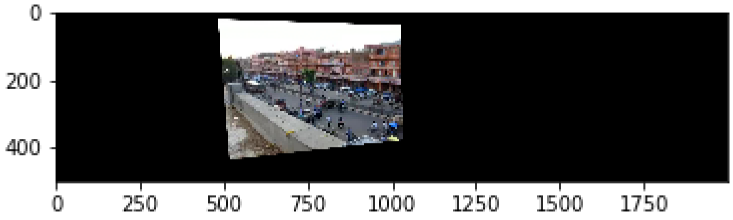


In [8]:

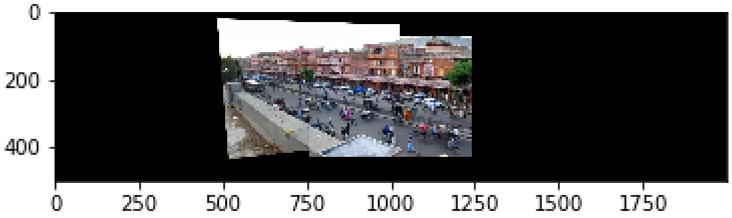
```
cols = 2000
rows = 500
# 360*480
# pts = np.float32([[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]]).reshape(-1, 1,
# dst = cv2.perspectiveTransform(pts, H)
H_transition = np.identity(3)
H_transition[0,2] = cols/2-240
H_transition[1,2] = rows/2-180
print(H_transition)
# Xb_ = np.dot(H, im1_plt) # project points from first image to second using
# du = Xb_[0,:]/Xb_[2,:]
# img_warped = cv2.warpPerspective(img, H_t.dot(H), (output_width, output_he

img_warped_270 = cv2.warpPerspective(im2_plt,H_transition,(cols, rows))
img_warped_0 = cv2.warpPerspective(im1_plt, np.dot(H_transition, H), (cols,
# img_warped3 = cv2.warpPerspective(img_warped2, H_transition, (w, h))

plt.imshow(img_warped_270)
plt.show()
plt.imshow(img_warped_0)
plt.show()
#  method 1
print("method1")
result = img_warped_270
for i in range(0,rows):
    for j in range(0,cols):
        if (result[i,j].sum() == 0) and (img_warped_0[i,j].sum() != 0):
            result[i,j] = img_warped_0[i,j]
plt.imshow(result)
plt.show()
```

```
[[   1.    0. 760.]
 [   0.    1.  70.]
 [   0.    0.   1.]]
```





```
method1
```



In [9]:

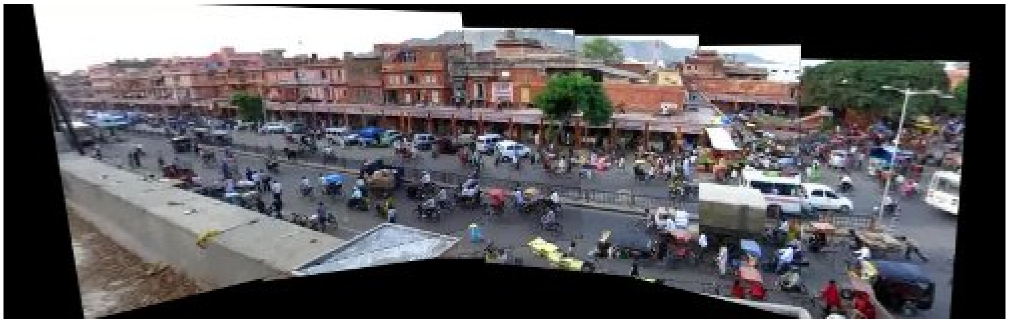
```
'''
#  method 2
print("method2")
for col in range(0, cols):
    if img_warped_270[:, col].any() and img_warped_0[:, col].any():
        left = col
        break
for col in range(cols-1, 0, -1):
    if img_warped_270[:, col].any() and img_warped_0[:, col].any():
        right = col
        break
srcImg = img_warped_270
warpImg = img_warped_0
res = np.zeros([rows, cols, 3], np.float32)

for row in range(0, rows):
    for col in range(0, cols):
        if not srcImg[row, col].any():
            res[row, col] = warpImg[row, col]
        elif not warpImg[row, col].any():
            res[row, col] = srcImg[row, col]
        else:
            srcImgLen = float(abs(col - right))
            testImgLen = float(abs(col - left))
            alpha = srcImgLen / (srcImgLen + testImgLen)
            res[row, col] = np.clip(srcImg[row, col] * (1-alpha) + warpImg[r
plt.imshow(res)
plt.show()
'''
```

Out[9]:

```
'\n#  method 2\nprint("method2")\nfor col in range(0, cols):\n    if img_
warped_270[:, col].any() and img_warped_0[:, col].any():\n        left =
col\n        break\nfor col in range(cols-1, 0, -1):\n    if img_warped_2
70[:, col].any() and img_warped_0[:, col].any():\n        right = col\n
break\nsrcImg = img_warped_270\nwarpImg = img_warped_0\nres = np.zeros([r
ows, cols, 3], np.float32)\n\nfor row in range(0, rows):\n    for col in
range(0, cols):\n        if not srcImg[row, col].any():\n            res
[row, col] = warpImg[row, col]\n        elif not warpImg[row, col].any
():\n            res[row, col] = srcImg[row, col]\n        else:\n
srcImgLen = float(abs(col - right))\n            testImgLen = float(abs(c
ol - left))\n            alpha = srcImgLen / (srcImgLen + testImgLen)\n
res[row, col] = np.clip(srcImg[row, col] * (1-alpha) + warpImg[row, col]
* alpha, 0, 255)\nplt.imshow(res)\nplt.show()\n'
```

Part II: Panorama using five key frames

In this part you will produce a panorama using five key frames. Let's determine frames [90, 270, 450, 630, 810] as key frames. The goal is to map all the five frames onto the plane corresponding to frame 450 (that we also call the *reference frame*). For the frames 270 and 630 you can follow the instructions in part 1.



Mapping frame 90 to frame 450 is difficult because they share very little area. Therefore you need to perform a two stage mapping by using frame 270 as a guide. Compute one projection from 90 to 270 and one from 270 to 450 and multiply the two matrices. This produces a projection from 90 to 450 even though these frames have very little area in common

In [10]:

```
import cv2
import numpy as np
import utils
```

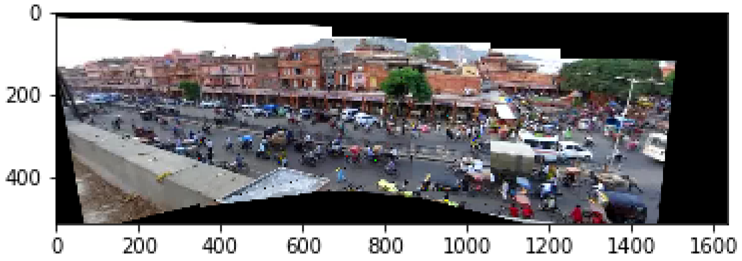
In [11]:

```
def warpped(result,img1,img2,H):
    # img2 = H*img1
    rows, cols, c = result.shape
#     cols = 1200
#     rows = 500
    img_warped = cv2.warpPerspective(img1, np.dot(H_transition, H), (cols, r
    return utils.blendImages(img_warped, result)
#     the following is written by myself instead of using utils.blendImages
#     for i in range(0,rows):
#         for j in range(0,cols):
#             if (result[i,j].sum() == 0) and (img_warped[i,j].sum() != 0):
#                 result[i,j] = img_warped[i,j]
#     return 0
```

In [12]:

```
master_frames =[90, 270, 450, 630, 810]
reference_frame = 450
reference_idx = master_frames.index(reference_frame)
im1_path = './images/input/frames/f0090.jpg'
im2_path = './images/input/frames/f0270.jpg'
im3_path = './images/input/frames/f0450.jpg'
im4_path = './images/input/frames/f0630.jpg'
im5_path = './images/input/frames/f0810.jpg'
im1_cv2 = cv2.imread(im1_path) # float32, 0.0-1.0
im2_cv2 = cv2.imread(im2_path)
im3_cv2 = cv2.imread(im3_path)
im4_cv2 = cv2.imread(im4_path)
im5_cv2 = cv2.imread(im5_path)

im1_RGB = cv2.cvtColor(im1_cv2,cv2.COLOR_BGR2RGB)
im2_RGB = cv2.cvtColor(im2_cv2,cv2.COLOR_BGR2RGB)
im3_RGB = cv2.cvtColor(im3_cv2,cv2.COLOR_BGR2RGB)
im4_RGB = cv2.cvtColor(im4_cv2,cv2.COLOR_BGR2RGB)
im5_RGB = cv2.cvtColor(im5_cv2,cv2.COLOR_BGR2RGB)

# fig, axes = plt.subplots(1,5)
# axes[0].imshow(im1_RGB)
# axes[1].imshow(im2_RGB)
# axes[2].imshow(im3_RGB)
# axes[3].imshow(im4_RGB)
# axes[4].imshow(im5_RGB)

H12 = auto_homography(im1_cv2,im2_cv2, computeHomography)
H23 = auto_homography(im2_cv2,im3_cv2, computeHomography)
H43 = auto_homography(im4_cv2,im3_cv2, computeHomography)
H54 = auto_homography(im5_cv2,im4_cv2, computeHomography)

H13 = np.dot(H23,H12)
H53 = np.dot(H43,H54)
cols = 1632
rows = 512
# 360*480
H_transition = np.identity(3)
H_transition[0,2] = cols/2-240
H_transition[1,2] = rows/2-180

result = cv2.warpPerspective(im1_RGB ,np.dot(H_transition, H13),(cols, rows)

result = warpped(result,im2_RGB,im3_RGB,H23)
result = warpped(result,im3_RGB,im3_RGB,np.identity(3))
result = warpped(result,im4_RGB,im3_RGB,H43)
result = warpped(result,im5_RGB,im3_RGB,H53)

# img_warped_0 = cv2.warpPerspective(im1_plt, np.dot(H_transition, H), (cols
# img_warped3 = cv2.warpPerspective(img_warped2, H_transition, (w, h))

plt.imshow(result)
plt.show()
```

```
best score: 208.000000
best score: 153.000000
best score: 132.000000
best score: 97.000000
```



Part 3: Map the video to the reference plane

In [13]:

```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from math import floor

# import utils
```

In [14]:

```python
dir_frames = 'images/input/frames'
filenames = []
filesinfo = os.scandir(dir_frames)
filesinfo
```

Out[14]:

```
<posix.ScandirIterator at 0x11b306ab0>
```

In [15]:

```python
filenames = [f.path for f in filesinfo if f.name.endswith(".jpg")]
filenames.sort(key=lambda f: int(''.join(filter(str.isdigit, f))))
filenames
```

Out[15]:

```
['images/input/frames/f0001.jpg',
 'images/input/frames/f0002.jpg',
 'images/input/frames/f0003.jpg',
 'images/input/frames/f0004.jpg',
 'images/input/frames/f0005.jpg',
 'images/input/frames/f0006.jpg',
 'images/input/frames/f0007.jpg',
 'images/input/frames/f0008.jpg',
 'images/input/frames/f0009.jpg',
 'images/input/frames/f0010.jpg',
 'images/input/frames/f0011.jpg',
 'images/input/frames/f0012.jpg',
 'images/input/frames/f0013.jpg',
 'images/input/frames/f0014.jpg',
 'images/input/frames/f0015.jpg',
 'images/input/frames/f0016.jpg',
 'images/input/frames/f0017.jpg',
 'images/input/frames/f0018.jpg',
 'images/input/frames/f0019.jpg',
```

In [16]:

```
frameCount = len(filenames)
frameHeight, frameWidth, frameChannels = cv2.imread(filenames[0]).shape
# rows, cols, 3 for a frame
frames = np.zeros((frameCount, frameHeight, frameWidth, frameChannels),dtype
# 900, rows, cols, 3 for all frames
```

In [17]:

```
# im2_cv2 = cv2.imread(im2_path)
# im2_RGB = cv2.cvtColor(im2_cv2,cv2.COLOR_BGR2RGB)
# im2_plt = im2_RGB.astype(np.float32)/255  !!! (value/255.0) instead of 25
for idx, file_i in enumerate(filenames):
    frames[idx] = cv2.cvtColor(cv2.imread(file_i), cv2.COLOR_BGR2RGB) / 255.
```

In [18]:

```
def auto_homography(Ia,Ib, homography_func=None, normalization_func=None):
    '''
    Computes a homography that maps points from Ia to Ib

    Input: Ia and Ib are images
    Output: H is the homography

    '''
    if Ia.dtype == 'float32' and Ib.dtype == 'float32':
        Ia = (Ia*255).astype(np.uint8)
        Ib = (Ib*255).astype(np.uint8)

    Ia_gray = cv2.cvtColor(Ia,cv2.COLOR_BGR2GRAY)
    Ib_gray = cv2.cvtColor(Ib,cv2.COLOR_BGR2GRAY)

    # Initiate SIFT detector
    sift = cv2.xfeatures2d.SIFT_create()

    # find the keypoints and descriptors with SIFT
    # kp_a is keypoints List and des_a is descriptors List
    kp_a, des_a = sift.detectAndCompute(Ia_gray,None)
    kp_b, des_b = sift.detectAndCompute(Ib_gray,None)

    # BFMatcher with default params
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des_a,des_b, k=2)
#     print('---TEST values in matches, which is list of elements(a,b)---')
#     print(matches[0])
#     print(matches[0][0].distance)
#     print(matches[0][1].distance)

    # Apply ratio test
    good = []
    for m,n in matches:
        if m.distance < 0.75*n.distance:
            good.append(m)

    numMatches = int(len(good)) # numMatches = 215
    matches = good

    # Xa and Xb are 3xN matrices that contain homogeneous coordinates for tl
    # matching points for each image
    Xa = np.ones((3,numMatches))
    Xb = np.ones((3,numMatches))

#     print('---TEST 3*215 point---')
#     print(Xa[:,0][0:2])
#     print(Xa[:,0])
    pts1_good = np.ones((3,4))
    pts2_good = np.ones((3,4))
    for idx, match_i in enumerate(matches):
        Xa[:,idx][0:2] = kp_a[match_i.queryIdx].pt
        Xb[:,idx][0:2] = kp_b[match_i.trainIdx].pt

    ## RANSAC
    niter = 1000
    best_score = 0

    H = np.zeros((3,3))
    for t in range(niter):
        # estimate homography
        subset = np.random.choice(numMatches, 4, replace=False)
        pts1 = Xa[:,subset]
        pts2 = Xb[:,subset]

        H_t = homography_func(pts1, pts2, normalization_func)
#         H_t = computeHomography(pts1, pts2) # edit helper code below (comp
        # pts1 = 3xN matrix and N = 4

        # score homography
        Xb_ = np.dot(H_t, Xa) # project points from first image to second us
        du = Xb_[0,:]/Xb_[2,:] - Xb[0,:]/Xb[2,:]
        dv = Xb_[1,:]/Xb_[2,:] - Xb[1,:]/Xb[2,:]

        ok_t = np.sqrt(du**2 + dv**2) < 1  # you may need to play with this 
        score_t = sum(ok_t)

        if score_t > best_score:
            best_score = score_t
            H = H_t
            in_idx = ok_t
            pts1_good = pts1
            pts2_good = pts2
#     print(pts1_good)
#     print(pts2_good)
    print('best score: {:02f}'.format(best_score))

# # Check that your homography is correct by plotting four points that form
#     # use as the four corners to draw the polylines
#     con_pts1 = np.zeros((4,2),np.int32)
#     con_pts2 = np.zeros((4,2),np.int32)

#     for i in range (4):
# #         print("point %d Xa"%(i),pts1_good[:,i])
# #         print("point %d Xb"%(i),pts2_good[:,i])
# #         Xb_ = np.dot(H, Xa)
# #         print("point %d Xb_"%(i),Xb_[:,i])
# #         print(Xb_[:,i]/Xb_[2,i])
#         con_pts1[i] = pts1_good[0:2,i]
#         con_pts2[i] = pts2_good[0:2,i]
# #         print(con_pts1[i])
# #         print(con_pts2[i])
# #     A[[i, j], :] = A[[j, i], :] # 实现了第i行与第j行的互换
#     im1_RGB2 = cv2.cvtColor(Ia,cv2.COLOR_BGR2RGB)
#     im2_RGB2 = cv2.cvtColor(Ib,cv2.COLOR_BGR2RGB)
#     con_pts1[[0, 2], :] = con_pts1[[2, 0], :]
#     con_pts2[[0, 2], :] = con_pts2[[2, 0], :]

# #   for draw rectangular  im1_RGB2 = cv2.rectangle(im1_RGB2,(100,100),(20
# #     pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
#     con_pts1 = con_pts1.reshape((-1,1,2))
#     con_pts2 = con_pts2.reshape((-1,1,2))
# #     print("--------")
# #     print(con_pts1)
# #     print(con_pts2)
#     im1_RGB2 = cv2.polylines(im1_RGB2,[con_pts1],True,(255,0,0),thickness
#     im2_RGB2 = cv2.polylines(im2_RGB2,[con_pts2],True,(255,0,0),thickness

#     plt.imshow(im1_RGB2)
#     plt.show()

#     plt.imshow(im2_RGB2)
#     plt.show()

#     # Optionally, you may want to re-estimate H based on inliers

    return H
```

In [19]:

```
def computeHomography(pts1, pts2, normalization_func=None):
    '''
    Compute homography that maps from pts1 to pts2 using least squares solve

    Input: pts1 and pts2 are 3xN matrices for N points in homogeneous
    coordinates.  N is 4.

    Output: H is a 3x3 matrix, such that pts2~=H*pts1
    '''

    dimen, num = pts1.shape
    A = np.zeros((2*num, 9), dtype = np.float32)

    for i in range (0, num):
        # p_prime = w*[u_prime,v_prime,1]
        # p = [u,v,1]
        p_prime = pts2[:,i]
        u_prime = p_prime[0]/p_prime[2]
        v_prime = p_prime[1]/p_prime[2]

        p = pts1[:,i]
        u = p[0]/p[2]
        v = p[1]/p[2]
        A[i*2,:] = [-u,-v,-1,0,0,0,u*u_prime, v*u_prime, u_prime]
        A[i*2+1,:] = [0,0,0,-u,-v,-1,u*v_prime, v*v_prime, v_prime]
    U,S,Vt = svd(A)
    h = Vt[-1,:]
    H = np.reshape(h, (3, 3))

#     std1_u = np.std(pts1[1,:])
#     std1_v =  np.std(pts1[2,:])
#     mean1_u = np.mean(pts1[1,:])
#     mean1_v = np.mean(pts1[2,:])
#     T1 = np.matrix([[1/std1_u, 0, 0],[0, 1/std1_v, 0],[0, 0, 1]]) * np.mat

#     std2_u = np.std(pts2[1,:])
#     std2_v = np.std(pts2[2,:])
#     mean2_u = np.mean(pts2[1,:])
#     mean2_v = np.mean(pts2[2,:])
#     T2 = np.matrix([[1/std2_u, 0, 0],[0, 1/std2_v, 0],[0, 0, 1]]) * np.mat
#     H = T2/H_n * T1
    return H
```

In [20]:
```
def projectImage(frames, sourceFrameIndex, referenceFrameIndex,
                 pastHomographies, originTranslations, xrange=2000,
                 yrange=800, overlapThreshold=40000, errorThreshold=4e-4,
                 numKeyframes=3, checkAllKeyframes=0, auto_H_func=None,
                 homography_func=None, normalization_func=None):
    '''
    Input:
        - frames: 4D array of frames
        - sourceFrameIndex: index of the frame to be projected
        - referenceFrameIndex: index of the frame to be projected to
        - pastHomographies: 2D cell array caching previously computed
          homographies from every frame to every other frame
        - xrange, yrange: dimensions of the output image
        - overlapThreshold: sufficient number of pixels overlapping between
          projected source and reference frames to ensure good homography
        - errorThreshold: acceptable error for good homography
        - numKeyframes: number of equidistant keyframes between source and
          reference frame to be visited in search of better homography
        - checkAllKeyframes: 0 if algorithms breaks after first better
          homography is found, 1 if all keyframes are to be visited

    Output:
        - bestProjectedImage: source frame optimally projected onto referenc
          frame using reestimation of homography based on closest-frame sear
          and using closest-frame homography as
    '''
        # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    H_transition = np.identity(3)
    H_transition[0,2] = 400 #cols
    H_transition[1,2] = yrange/2-180 #rows
#     360 * 480
    print("once")
    numFrames = frames.shape[0]
    _, referenceTransform, ref_origin_coord = transformImage(frames, referen
    _, sourceTransform, src_origin_coord = transformImage(frames, sourceFran
    _, err = computeOverlap(sourceTransform, src_origin_coord, referenceTran
    originTranslations[sourceFrameIndex] = src_origin_coord


    x_min, y_min = originTranslations[0]
    # Translation matrix
    t = [-x_min, -y_min]
    H_t = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]], dtype=np.float32
    # Dot product of translation matrix and homography
    pastHomographies[sourceFrameIndex, referenceFrameIndex] = H_t.dot(pastHo

#     projectedImage = cv2.warpPerspective((frames[sourceFrameIndex]*255).as
#                                          pastHomographies[sourceFrameIndex
#                                          (xrange, yrange))
    projectedImage = cv2.warpPerspective((frames[sourceFrameIndex]*255).asty
                                         H_transition.dot(pastHomographies[s
                                         (xrange, yrange))



    if err > errorThreshold:
        print('Finding better homography...')
        increment = floor(((referenceFrameIndex - sourceFrameIndex) - 1) / (
        keyframeIndex = sourceFrameIndex + increment  # frame being used to
        found = 0
        counter = 0
        bestHomography = np.eye(3)  # initialize H as identity

        while counter < numKeyframes and keyframeIndex < numFrames and keyfr

            # compute homography and projected image from keyframe to
            # reference frame
            H2, keyframeTransform, keyframe_origin_coord = transformImage(fr
            a, error1 = computeOverlap(keyframeTransform, keyframe_origin_co

            # compute homography and projected image from source frame to
            # keyframe (new reference = keyframe)
            _, keyframeToKeyframeTransform, keyframeToKeyframe_origin_coord
            H1, sourceToKeyframeTransform, srcToKeyframe_origin_coord = tran
            b, error2 = computeOverlap(sourceToKeyframeTransform, srcToKeyfr

            sufficientOverlap = (a and b)

            if (sufficientOverlap and max(error1, error2) < err):
                found = 1
                bestHomography = np.dot(H1, H2)
                src_origin_coord = keyframe_origin_coord + srcToKeyframe_ori
                if not checkAllKeyframes:
                    break

            keyframeIndex = keyframeIndex + increment
            counter = counter + 1

        if found:
            print('Found better homography')
            pastHomographies[sourceFrameIndex, referenceFrameIndex] = bestHo
            originTranslations[sourceFrameIndex] = src_origin_coord
            min_origin_coord = np.amin(originTranslations, axis=0)

            x_min, y_min = originTranslations[0]
            # Translation matrix
            t = [-x_min, -y_min]
            H_t = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]], dtype=np

            # Dot product of translation matrix and homography
            T = H_t.dot(bestHomography)

#             projectedImage = cv2.warpPerspective((frames[sourceFrameIndex]

            projectedImage = cv2.warpPerspective((frames[sourceFrameIndex]*2

            pastHomographies[sourceFrameIndex, referenceFrameIndex] = T.asty

    return projectedImage, pastHomographies, originTranslations
```

In [21]:

```
## Example usage of utils.projectImage

pastHomographies = np.zeros((len(filenames),len(filenames), 3, 3),dtype=np.f
# 900*900*(3*3) H13 = pastHomographies[1,3]
originTranslations = np.zeros((len(filenames), 2), dtype=np.float32)
# store the current homographies
# sourceFrameIndex = 630
# referenceFrameIndex = 450
```

In [22]:

```
from utils import transformImage, computeOverlap
```

In [23]:

```
# frameCount = len(filenames)
# frameHeight, frameWidth, frameChannels = cv2.imread(filenames[0]).shape
# # rows, cols, 3 for a frame
# frames = np.zeros((frameCount, frameHeight, frameWidth, frameChannels),dty
# # 900, rows, cols, 3 for all frames

# idx start from 0
# for idx, file_i in enumerate(filenames):
#     frames[idx] = cv2.cvtColor(cv2.imread(file_i), cv2.COLOR_BGR2RGB) / 25

pastHomographies = np.zeros((len(filenames),len(filenames), 3, 3),dtype=np.f
# 900*900*(3*3) H13 = pastHomographies[1,3]
originTranslations = np.zeros((len(filenames), 2),dtype=np.float32)
```

In [24]:

```
# blendOutput_images = np.zeros((900,800,2000,3))
# 900 * rows * cols * 3
master_frames =[x for x in range (0,len(filenames))]

reference_frame = 450
referenceFrameIndex = master_frames.index(reference_frame)
print(referenceFrameIndex)

projectedSource, pastHomographies, originTranslations = projectImage(frames,
                                                            pastHomographies
                                                            auto_H_func=auto


H_transition = np.identity(3)
H_transition[0,2] = 400 #cols
H_transition[1,2] = 800/2-180 #rows
# H = np.identity(3)
# projectedSource = cv2.warpPerspective((frames[450]*255).astype(np.uint8),
projectedSource = cv2.cvtColor(projectedSource,cv2.COLOR_RGB2BGR)
cv2.imwrite('images/input/aligned_frames/a{:04d}.jpg'.format(referenceFrameI
print('------frame{:04d} have been written------'.format(450))
# blendOutput_images[referenceFrameIndex] = projectedSource

for i in range(0,499):
    sourceFrameIndex = master_frames[i]

    projectedSource, pastHomographies, originTranslations = projectImage(fra
                                                            pastHomographies
                                                            auto_H_func=auto
    projectedSource = cv2.cvtColor(projectedSource,cv2.COLOR_RGB2BGR)
    cv2.imwrite('images/input/aligned_frames/a{:04d}.jpg'.format(i), project
    print('------frame{:04d} have been written------'.format(i))
#     blendOutput_images[sourceFrameIndex] = projectedSource

for i in range(899,450,-1):
    print(i)
#     projectedSource = cv2.warpPerspective((frames[sourceFrameIndex]*255).a
#                                           H_transition.dot(auto_homography(
#                                           (2000, 800))
    sourceFrameIndex = master_frames[i]

    projectedSource, pastHomographies, originTranslations = projectImage(fra
                                                            pastHomographies
                                                            auto_H_func=auto
    projectedSource = cv2.cvtColor(projectedSource,cv2.COLOR_RGB2BGR)
    cv2.imwrite('images/input/aligned_frames/a{:04d}.jpg'.format(i), project
    print('------frame{:04d} have been written------'.format(i))
#     blendOutput_images[sourceFrameIndex] = projectedSource
```

```
Overlap:40269
Error:0.0005286348931498656
best score: 2753.000000
best score: 277.000000
Overlap:147990
Error:0.00018461976431996416
Found better homography
------frame0001 have been written------
once
```

```
best score: 34.000000
Overlap:24415
Error:0.0007139659090422842
Finding better homography...
Overlap:40269
Error:0.0005286348931498656
best score: 278.000000
Overlap:148610
Error:0.00018448426329937 47
Found better homography
------frame0002 have been written------
```

In [25]:

```
utils.imageFolder2mpeg('./images/input/aligned_frames_100',output_path='./i
```

```
---------------------------------------------------------------------------
--
FileNotFoundError                         Traceback (most recent call las
t)
<ipython-input-25-525fea9713cf> in <module>()
----> 1 utils.imageFolder2mpeg('./images/input/aligned_frames_2',output_p
ath='./images/input/output_video_result.mpeg', fps=30.0)

~/Desktop/CS 445/CS445_proj5/Project 5/jieting2_proj5/utils.py in imageFo
lder2mpeg(input_path, output_path, fps)
    290
    291     dir_frames = input_path
--> 292     files_info = os.scandir(dir_frames)
    293
    294     file_names = [f.path for f in files_info if f.name.endswith(
".jpg")]

FileNotFoundError: [Errno 2] No such file or directory: './images/input/a
ligned_frames_2'
```

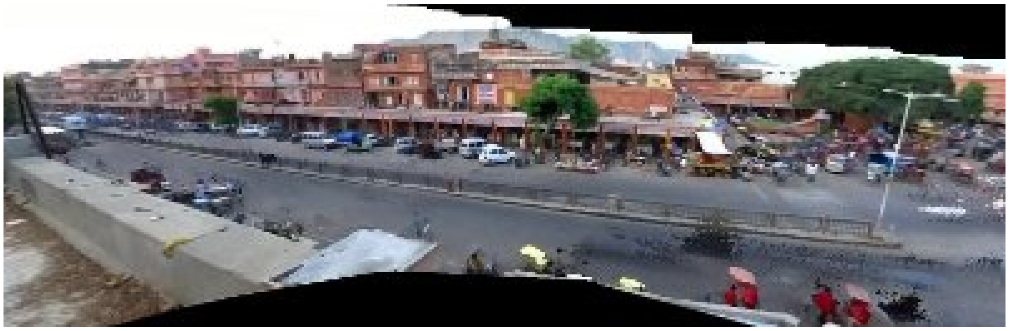
In [ ]:

```
# im2_cv2 = cv2.imread(im2_path)
# im2_RGB = cv2.cvtColor(im2_cv2,cv2.COLOR_BGR2RGB)
# im2_plt = im2_RGB.astype(np.float32)/255  !!! (value/255.0) instead of 25
# for idx, file_i in enumerate(filenames):
#     frames[idx] = cv2.cvtColor(cv2.imread(file_i), cv2.COLOR_BGR2RGB) / 2
```

Part 4: Create background panorama

In this part you will remove moving objects from the video and create a background panorama that should incorporate pixels from all the frames.

In the video you produced in **part 3** each pixel appears in several frames. You need to estimate which of the many colors correspond to the background. We take advantage of the fact that the background color is fixed while the foreground color changes frequently (because foreground moves).



For each pixel in the sequence of **part 3**, determine all valid colors (colors that come from all frames that overlap that pixel). You can experiment with different methods for determining the background color of each pixel, as discussed in class. Perform the same procedure for all pixels and generate output. The output should be a completed panorama showing only pixels of background or non-moving objects.

In [26]:

```
import utils
import cv2
import numpy as np
from numpy.linalg import svd, inv

%matplotlib inline
from matplotlib import pyplot as plt
import os

from math import floor

# import utils
```

In [27]:

```
#  read files from the wrapped images
dir_frames_transfered = 'images/input/frames_transfered'
wrapped_imgs = []
filesinfo_transfered = os.scandir(dir_frames_transfered)
filesinfo_transfered
```

Out[27]:

```
<posix.ScandirIterator at 0x11b447750>
```

In [28]:

```
filenames_transfered = [f.path for f in filesinfo_transfered if f.name.endsw
filenames_transfered.sort(key=lambda f: int(''.join(filter(str.isdigit, f)))
filenames_transfered
```

```
 'images/input/frames_transfered/a0066.jpg',
 'images/input/frames_transfered/a0067.jpg',
 'images/input/frames_transfered/a0068.jpg',
 'images/input/frames_transfered/a0069.jpg',
 'images/input/frames_transfered/a0070.jpg',
 'images/input/frames_transfered/a0071.jpg',
 'images/input/frames_transfered/a0072.jpg',
 'images/input/frames_transfered/a0073.jpg',
 'images/input/frames_transfered/a0074.jpg',
 'images/input/frames_transfered/a0075.jpg',
 'images/input/frames_transfered/a0076.jpg',
 'images/input/frames_transfered/a0077.jpg',
 'images/input/frames_transfered/a0078.jpg',
 'images/input/frames_transfered/a0079.jpg',
 'images/input/frames_transfered/a0080.jpg',
 'images/input/frames_transfered/a0081.jpg',
 'images/input/frames_transfered/a0082.jpg',
 'images/input/frames_transfered/a0083.jpg',
 'images/input/frames_transfered/a0084.jpg',
 'images/input/frames_transfered/a0085.jpg',
```

In [29]:

```
Count = len(filenames_transfered)
Height, Width, Channels = cv2.imread(filenames_transfered[0]).shape
# rows, cols, 3 for a frame
wrapped_imgs = np.zeros((Count, Height, Width, Channels),dtype=np.float32)

for idx, file_i in enumerate(filenames_transfered):
    wrapped_imgs[idx] = cv2.cvtColor(cv2.imread(file_i), cv2.COLOR_BGR2RGB)
```

In [30]:

```
print(wrapped_imgs[1,450,400])
```

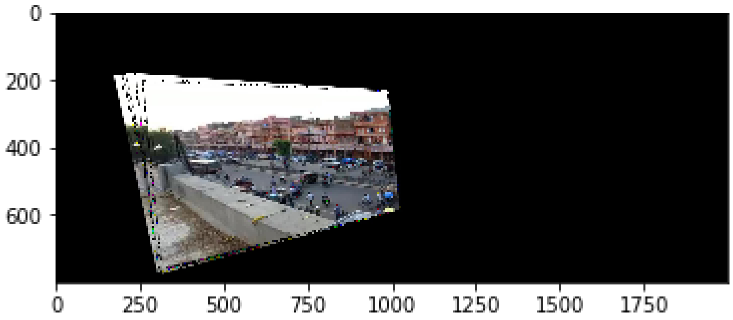
```
[0.16862746 0.16862746 0.20784314]
```

In [31]:

```
def my_blendImages(sourceTransform, referenceTransform):
    '''
    Input:
        - sourceTransform: source frame projected onto reference frame plane
        - referenceTransform: reference frame projected onto same space

    Output:
        - blendedOutput: naive blending result from frame stitching
    '''

    blendedOutput = referenceTransform
    indices = referenceTransform == 0.0
    blendedOutput[indices] = sourceTransform[indices]

    return blendedOutput
#     return (blendedOutput / blendedOutput.max() * 255).astype(np.uint8)
```

In [32]:

```
result = np.zeros((Height, Width, Channels),dtype = np.float32)
for i in range (0, 10,1):
    result1 = np.zeros((Height, Width, Channels),dtype = np.float32)
    result1 = my_blendImages(wrapped_imgs[i],result)
    result = result1
# warpped2(wrapped_imgs[0],result)
# warpped2(wrapped_imgs[200],result)
# warpped2(wrapped_imgs[400],result)
# warpped2(wrapped_imgs[600],result)
# warpped2(wrapped_imgs[800],result)
plt.imshow(result)
plt.show()


# for i in range (0,Count):
#     result = utils.blendImages(wrapped_imgs[i], result)
# # blendedOutput = utils.blendImages(projectedSource, projectedReference)

# plt.imshow(result)
# plt.show()
```



In [ ]:

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import statistics
import pdb

# for each pixel, find the median across the 900 frames
# vectorized approach
rows,cols = wrapped_imgs[0].shape[0], wrapped_imgs[0].shape[1]
median_calc = np.zeros((rows,cols,3,len(wrapped_imgs)))
# 800 * 2000 * 3 * 900
median_calc.fill(np.nan)

frame,x,y,c = np.nonzero(wrapped_imgs)


median_calc[x,y,c,frame] = wrapped_imgs[frame,x,y,c]
median = np.nanmedian(median_calc,axis=3) # Compute the median along the spe
```

In [ ]:
```
plt.figure(figsize=(65,15))
plt.imshow(median/255)
```

Part 5: Create background movie

Map the background panorama to the movie coordinates. For each frame of the movie, say frame 1, you need to estimate a projection from the panorama to frame 1. Note, you should be able to re-use the homographies that you estimated in **Part 3**. Perform this for all frames and generate a movie that looks like the input movie but shows only background pixels. All moving objects that belong to the foreground must be removed.

In [ ]:
```
import os
import cv2
import numpy as np
```

In [ ]:
```
for i in range(0, 2):
    cur_H = pastHomographies[i][450]
    inv_H = np.linalg.inv(cur_H)
    img_warped = cv2.warpPerspective(median, inv_H, (360, 480))
```

In [ ]:
```
utils.imageFolder2mpeg('./images/input/background_frame',output_path='./imag
```

Part 6: Create foreground movie

In the background video, moving objects are removed. In each frame, those pixels that are different enough than the background color are considered foreground. For each frame determine foreground pixels and generate a movie that only includes foreground pixels.

In [ ]:
```
import os
import cv2
import numpy as np

for i in range(0, Count):
    difference = frames[i] - (warped_imgs[i]/255)
    cv2.imwrite('images/input/foreground_frames/a{:04d}.jpg'.format(i), diffe
utils.imageFolder2mpeg('./images/input/foreground_frames',output_path='./imag
```

Bells and whistles

In [ ]:
```
# in Part 1 and file my_example1.ipynb and my_example2.ipynb
```